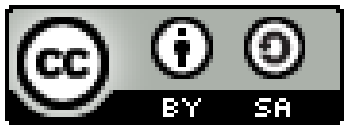
Нити выполнения

POSIX pthreads

pthreads или управляемая шизофрения



Процессы как действия
Сильные стороны

Операционная системы обеспечивает процессы множеством полезных вещей:

- **Защита** — все процессы имеют своё независимое адресное пространство — образ процесса надёжно защищён от доступа извне. И наоборот — попытки процесса получить доступ к чужой памяти пресекаются.
- **Надёжное планирование** — планировщик задач выделяет процессорное время всем процессам (по выбранной стратегии), как бы процессы не пытались обделить соседей по ОС.
- **Виртуальная память** — мощные и простые средства управления адресным пространством.
- **Средства IPC** — множество средств межпроцессного взаимодействия на любой вкус.

Процессы как действия
Слабые стороны

Существуют задачи, в который над одними и теми же данными выполняются несколько действий. Если каждое действие оформить в виде процесса, а данные передавать через один из механизмов IPC, то сталкиваемся с такими отрицательными явлениями:

- Переключение между процессами — достаточно сложное и длительное действие. Если часто переключаться — большие потери производительности.
- Практически каждый метод IPC требует накладных расходов.
- Существуем ограничение по количеству процессов, как на пользователя, так и по системе в целом.

Можно в рамках одного процесса создать подобие планировщика — но это сложная задача, и требующая (в большинстве случаев) поддержки со стороны аппаратного обеспечения.

Нити выполнения
Несколько потоков выполнения кода

В рамках одного процесса может выполняться одновременно или почти одновременно несколько нитей выполнения (threads) — последовательностей команд процессора.

Сегменты кода, данных, стека — **общие**, но каждая нить имеет **свой участок** в стеке, где хранятся локальные переменные.

Общие PID, PPID, PGID, *UID, *GID, umask, текущий каталог. Файлы — общие.

Свои наборы сигналов и сигнальная маска. Своя переменная *errno*. Некоторые функции — безопасные для нитей (thread-safe), некоторые — нет.

Сборка программ с поддержкой нитей

Требуются специальные действия для правильной сборки программ и использованием нитей.

1. В файлы исходного кода включить заголовок

```
#include <pthread.h>
```

2. Подключить библиотеку **libpthread**, например задав в Makefile

```
LDFLAGS=-lpthread
```

3. Передать ключ “-pthread” компилятору:

```
CFLAGS=-pthread -Wall -g3 -O2
```

Создание нити

Нить создаётся с помощью функции:

```
int pthread_create( pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine) (void *),
        void *arg);
```

pthread_t *thread — указатель на переменную, в которую будет записан идентификатор новой нити.
const pthread_attr_t *attr — указатель на структуру атрибутов создаваемой нити.
Если 0 — создаётся нить по умолчанию (joinable, . . .).
void *(*start_routine) (void *) — указатель на функцию, с которой начинается выполнение нити. Функция получает void* и возвращает void*. Следует обеспечить существование переменных, на которые передаются указатели.
void* arg — значение аргумента, которое передаётся функции.

Создание нити. 2

Возвращает:
0 – Ok. !=0 – номер ошибки. errno НЕ используется. Так ведут себя большинство функция для работы с нитями выполнения.

Нитью должен быть обособленный участок кода. В C, C++ и подобных языках — это функции. Т.е выполнение нити начинается с заданной функции, и завершается либо вместе с завершением функции, либо досрочно (pthread_exit, pthread_cancel).

После удачного вызова pthread_create в рамках процесса становиться на одну нить выполнения больше.
Отношения родитель-потомок не используются.
У каждой нити — свой идентификатор.

Завершение нити

Если надо завершить нить из той функции, с которой началось её выполнение, можно использовать инструкцию **return** :

```
return some_ptr;
```

В любом другом месте следует использовать функцию

```
void pthread_exit(void *retval);
```

void* retval — значение, которое будет получено тем, кто ждет завершения этой нити (если она joinable) или будет утеряно (если она datached).

Никогда не возвращает значения (некуда).

Если в любой нити вызвать exit, то завершаться сразу все нити.

Присоединяемые и отсоединённые нити

По умолчанию нити **присоединяемые** (joinable), т.е. любая другая нить этого же процесса дождаться её завершения и получить возвращаемое значение.

Нить может быть **отсоединённой** (detached). В этом случае никто не может и не должен ждать её завершения. Нить можно или создать отсоединённой (используя структуру атрибутов), а можно отсоединить потом:

```
int pthread_detach(pthread_t thread);
```

pthread_t thread — идентификатор отсоединяемой нити (можно себя).

Ожидание завершения нити

Дождаться завершения заданной нити и присоединить ее можно с помощью функции

```
int pthread_join(pthread_t thread, void **retval);
```

pthread_t thread — идентификатор присоединяемой нити, завершения которой ожидается (себя нельзя).

void **retval — указатель на переменную, в которую будет записано возвращаемое значение нити.

Возвращает: 0 — Ok ; !=0 — код ошибки

Снятия нити с выполнения
Cancellation

Снять нить с выполнения можно с помощью функции

```
int pthread_cancel(pthread_t thread);
```

pthread_t thread — идентификатор нити, которую надо снять (принудительно завершить).

Завершение происходит только в точках отмены (cancellation point) Можно запретить/разрешить с помощью pthread_setcancelstate(3)

Работа с идентификаторами нитей

Узнать свой идентификатор нити;

```
pthread_t pthread_self(void);
```

– возвращает идентификатор текущей нити

Сравнить на равенство два идентификатора

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

– возвращает !=0 , если t1 и t2 соответствуют одной и той же нити.

Работа с атрибутами

Для хранения атрибутов создаваемых нитей используется структура **pthread_attr_t** . Внутренний формат — детали реализации, и многократно изменялся. Поэтому для переносимой работы используется набор функций.

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Инициализация и деинициализация структуры. После инициализации структуру можно использовать для создания нитей по умолчанию.

Работа с атрибутами

Признак состояния “detached”

Для уменьшения вероятности появления “race condition” (состояния гонок) отсоединённые нити предпочтительно создавать сразу отсоединёнными.

```
int pthread_attr_setdetachstate(pthread_attr_t *at,
      int detachstate);
int pthread_attr_getdetachstate(pthread_attr_t *at,
      int *detachstate);
 // flags
 // PTHREAD_CREATE_DETACHED --- отсоединённая нить
 // PTHREAD_CREATE_JOINABLE --- присоединяемая
```

Все функции из этого блока начинаются с “pthread_attr_”, потом set или get – соответственно установить или узнать состояние требуемого параметра. Следующая часть имени функции символизирует имя параметра.

Работа с атрибутами

Выбор режима планирования

Если заранее известны специальные запросы нити на планирование, можно настроить политику и параметры планировщика.

```
int pthread_attr_setschedpolicy(pthread_attr_t *at,
    int policy);
int pthread_attr_getschedpolicy(pthread_attr_t *at,
    int *policy);
// SCHED_OTHER   стандартная политика планирования
// SCHED_BATCH   политика для фоновых процессов
// SCHED_IDLE    ... с низким приоритетом.
// ------ REALTIME:
// SCHED_FIFO    a first-in, first-out policy
// SCHED_RR  a round-robin policy.
```

Работа с атрибутами
Параметры планирования

```
int pthread_attr_setschedparam(pthread_attr_t *at,
        const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *at,
struct sched_param *param);

struct sched_param {
  int sched_priority;     /* Scheduling priority */
};
```

Работа с атрибутами

Дополнительные параметры

```
int pthread_attr_setguardsize(pthread_attr_t *at,
      size_t guardsize);
int pthread_attr_getguardsize(pthread_attr_t *at,
     size_t *guardsize);
int pthread_attr_setinheritsched(pthread_attr_t *at,
      int inheritsched);
int pthread_attr_getinheritsched(pthread_attr_t *at,
      int *inheritsched);
int pthread_attr_setscope(pthread_attr_t *attr,
      int scope);
int pthread_attr_getscope(pthread_attr_t *attr,
       int *scope);
int pthread_attr_setstack(pthread_attr_t *attr,
       void *stackaddr, size_t stacksize);
int pthread_attr_getstack(pthread_attr_t *attr,
       void **stackaddr, size_t *stacksize);
// .......
```

Мьютексы

Средства синхронизации

Монопольный доступ к ресурсам

Проблема одновременного доступа к ресурсам

При наличии более одной нити выполнения в рамках одного процесса возможен **одновременный** доступ нескольких нитей к одному общему ресурсу.

- Структура данных (списки, деревья, очереди . . .).
- Буферы файлового обмена
- Элементы интерфейса пользователя.
- Средства управления оборудованием.

Если существуют только процессы, то большинство задач синхронизации ОС берёт на себя. При нескольких нитях выполнения все проблемы — это проблемы программиста.

Организация монопольного доступа к общим ресурсам.

Для организации монопольного доступа к общим ресурсам (памяти, потокам ввода-вывода, аппаратному обеспечению) нужен механизм (аналогичный замку), который может находиться в двух состояниях:

1. свободен (открыто)
2. занято (закрыто).

Причем операционная система должна обеспечивать своими средствами, аппаратными (хорошо) и программными (допустимо), что к самому замку будет одновременный доступ только одному субъекту (нити или процессу). Если какая-то нить захватила ресурс (закрыла замок), то другая при попытке захвата будет заблокирована до тех пор, пока первая не снимет блокировку.

Мьютексы (mutex)

В стандарте POSIX указано, что для этих целей должны быть предоставлены мьютексы (mutex — **mu**tual **ex**clusive).
Описаны в том же заголовочном файле, что и нити выполнения:

```
#include <pthread.h>
```

Сам мьютекс — это объект (переменная) типа pthread_mutex_t:

```
pthread_mutex_t lock;
```

Её надо разместить в месте, доступном для всех участников и инициализировать (функцией или в момент создания).

Инициализация мьютекса
Инициализация при создании

Проще всего мьютекс инициализировать в момент создания:

```
pthread_mutex_t mutex =  PTHREAD_MUTEX_INITIALIZER;
 // обычный

 // PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP
 // – рекурсивный есть( счетчик блокировок)

 //  PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP
 //  – с контролем хозяина.
```

Можно использовать только сам мьютекс, нельзя — его копию.

Инициализация мьютекса
Инициализация функциями

Если нет возможности статической инициализации, можно и с помощью функций:

```
int pthread_mutex_init(
    pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Мьютексы — одноразовые, нельзя повторно инициализировать освобождённый.

Настройка атрибутов мьютексов

```
int pthread_mutexattr_init(pthread_mutexattr_t *at);
   // инициализация
int pthread_mutexattr_destroy(
   pthread_mutexattr_t *attr);
  // освобождение
int pthread_mutexattr_gettype (
    const pthread_mutexattr_t *__restrict attr,
          int *__restrict kind); // получить тип
int pthread_mutexattr_settype (
       pthread_mutexattr_t *attr,  int kind);
     // задать тип:

// PTHREAD_MUTEX_NORMAL,
// PTHREAD_MUTEX_RECURSIVE,
// PTHREAD_MUTEX_ERRORCHECK,
// PTHREAD_MUTEX_DEFAULT
```

Блокировка мьютекса

Когда надо захватить ресурс, ее блокируют:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
 // 0 - Ok !=0 - error
```

Если mutex не был заблокирован, нить получает блокировку и продолжает работать дальше.
В противном случае - нить блокируется до тех пор, пока кто-то не снимет блокировку с этого мьютекса. При этом нить разблокируется, сама получает блокировку и продолжает работать.

Снятие блокировки

Когда ресурс больше не нужен, соответствующий mutex разблокируют:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
  // 0 - Ok !=0 - error
```

Проверка занятости

Есть еще функция, аналогичная pthread_mutex_lock, но если ресурс занят, но функция не блокируется, а возвращается ошибка EBUSY.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
  // 0 - Ok !=0 - error
```

Типы мьютексов

Если нить с обычным мьютексом попытается его заблокировать повторно, то она заблокирует себя до своей смерти.

Если нити разные, то рекурсивные mutex-ы работают как обычные. Если одна и та же нить будет блокировать рекурсивный mutex, то просто увеличится счетчик блокировок. Разблокировать в этом случае надо столько раз, сколько блокировали.

Если используется mutex с проверкой ошибок, то при попытке повторно заблокировать свой mutex функция pthread_mutex_lock вернет ошибку, и самоблокировки не будет.

Современные расширения

Захват с ограничением по времени:

```
int pthread_mutex_timedlock(
        pthread_mutex_t *restrict mutex,
        const struct timespec *restrict abs_timeout);
```

Мьютексы чтения-записи

```
int pthread_rwlock_init(
    pthread_rwlock_t *restrict rwlock,
    const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

int pthread_rwlock_rdlock(
    pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(
    pthread_rwlock_t *rwlock);

int pthread_rwlock_wrlock(
    pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(
    pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(
    pthread_rwlock_t *rwlock);
```

Другие средства

Мьютексы считаются нижним уровнем систем организации монопольного доступа и синхронизации. С их помощью реализуют другие средства:

- условные переменные (pthread_cond_t);
- семафоры
- барьеры
- spin-lock
- ...